Memory-Reference Characteristics of Multiprocessor Applications under MACH

Anant Agarwal* and Anoop Gupta
Computer Systems Laboratory
Stanford University, CA 94305

1988

Abstract

Shared-memory multiprocessors have received wide attention in recent times as a means of achieving high-performance cost-effectively. Their viability requires a thorough understanding of the memory access patterns of parallel processing applications and operating systems. This paper reports on the memory reference behavior of several parallel applications running under the MACH operating system on a shared-memory multiprocessor. The data used for this study is derived from multiprocessor address traces obtained from an extended ATUM address tracing scheme implemented on a 4-CPU DEC VAX 8350. The applications include parallel OPS5, logic simulation, and a VSLI wire routing program. Among the important issues addressed in this paper are the amount of sharing in user programs and in the operating system, comparing the characteristics of user and system reference patterns, sharing related to process migration, and the temporal, spatial, and processor locality of shared blocks. We also analyze the impact of shared references on cache coherence in shared-memory multiprocessors.

1 Introduction

Although we now have a reasonably good understanding of memory system design for uniprocessors, very little is understood about memory system design for multiprocessors. A major reason for this has been the lack of real data about memory reference patterns for multiprocessors, because of the difficulty of tracing such machines. The problem of getting realistic trace data is even more acute if one wishes to the study the effects of operating system references, process migration, and other such real system events. This paper attempts to correct this situation and analyzes memory reference patterns of several parallel applications running under the MACH operating system on a shared-memory multiprocessor. The address traces used in our study were obtained from a 4-processor VAX 8350 multiprocessor using an extended version of the ATUM [1] address tracing technique. These traces contain both system and user memory references, including process migration information.

Analysis of shared-memory reference patterns is needed to determine the most suitable organization of the memory hierarchy in multiprocessors. For example, several cache consistency algorithms proposed in the literature are based on subtle differences in the expected memory reference patterns; lacking detailed data, the benefits of one scheme over another cannot be assessed accurately. While some previous studies have looked at shared-memory reference patterns, e.g., [2], they did not fully address issues such as the temporal, spatial, and processor locality of shared data, sharing in the operating system, and the impact on cache consistency. For example, we show that shared references display a significant amount of processor locality. The average number of read and write references to a write-shared block before a remote reference are 4 and 2 respectively. This locality is exploited by the write-back class of cache coherence schemes to significantly reduce the cost of references to shared data. Another surprising result that we observed for shared data references is that the total bus bandwidth required is minimized when block size is 4 bytes and increases as the block size is increased. We also observe that processor migration causes a large increase in the sharing level as observed by the caches, which can greatly increase cache coherence traffic on the bus.

This paper is organized as follows. Section 2 presents background information about the ATUM address tracing technique, the applications measured, and the MACH operating system. Section 3 defines our multiprocessor model and the terminology used throughout the paper. Section 4 constitutes the bulk of the paper and is devoted to analyzing the traces. This section characterizes shared-memory reference patterns and looks at the impact of the reference characteristics on cache consistency algorithms. Specifically, in Section 4.1 we present data about the general characteristics of the traces, including statistics about interlocked instructions. Section 4.2 assesses the temporal and processor locality of shared references. Section 4.3 focuses on how the memory reference characteristics affect the performance of various cache consistency algorithms. Section 5 concludes the paper.

2 Background and Methodology

Our study is based on trace analysis. The traces are obtained using a multiprocessor extension of the ATUM tracing scheme [1]. ATUM stands for Address Tracing Using Microcode and works as follows: During the execution of each instruction, the microcode writes out the memory references

*Anant Agarwal is currently with the Laboratory for Computer Science (NE43-418), M.I.T. Cambridge, MA 02139 made by the processor to a portion of memory reserved for tracing. In the multiprocessor extension of ATUM, each access to trace memory is interlocked to enable the microcode in several processors to write their references to this memory. Thus a trace contains interleaved address streams of several processors. The traces used for this study were gathered on a 4-CPU VAX 8350 machine running the MACH operating system. ATUM traces are "complete" in that they capture all operating system and multiprogramming activity. Each trace is roughly 3.5 million references long. In addition to addresses, ATUM records the opcodes, and the virtual-to-physical translations that occur during translation-lookaside-buffer (TLB or TB) misses. A location is considered shared when it is referenced by more than one CPU. Because different processes could access a given shared location with different virtual addresses, sharing is detected by translating the various virtual addresses of a shared location to its common physical address.

The traces used in this paper are obtained from three programs: POPS, THOR, and PERO. POPS [3] is a parallel implementation of a rule-based programming language called OPS5, which is a widely used languages for the building expert systems. It exploits parallelism at a fine granularity and makes extensive use of the shared memory provided by the architecture. THOR is a parallel implementation of a logic simulator done by Larry Soule at Stanford University. The simulator transforms the task of circuit simulation into a series of node evaluations, where each node corresponds to a device in the circuit. The parallel implementation evaluates these nodes in parallel, while taking care of the dependencies between them. PERO is a parallel VLSI router written by Jonathan Rose at Stanford [4].

We briefly describe the MACH operating system, since some of the shared references in the traces belong to it, and also because the programming style used in the applications was influenced by it. MACH is a multiprocessor operating system developed at Carnegie Mellon University. It is binary compatible with Berkeley Unix, and provides several new facilities to support parallel processing. It provides facilities for multiple tasks to share memory permitting the exploitation of very fine grained parallelism. All three programs make use of multiple tasks that share memory to communicate with each other and to share information. MACH is not a totally symmetric operating system in that kernel interrupts are handled by processor zero. This causes the memory reference pattern of processor zero to be different from that of the remaining processors. In parallel programs, where many tasks are performing I/O, the high level of OS interrupts can also cause excessive process migration. Fortunately, none of the programs that we study in this paper do very much I/O.

Table 1 presents general trace characteristics for the three programs. The columns denote the total number of references, instruction references, data reads, data writes, user and system references. Instruction and data references are about equal, while there are roughly three reads to every write. About 12% of all references are system.

The ATUM traces used for this study do have some limitations. The machine used had only 4 CPUs and it is not clear how to extend the results to a larger number of processors. Work on this issue is in progress. Another problem is the unavailability of a large number of applications, but the number is growing.

Table 1: Summary of trace characteristics. All numbers are in thousands.

Trace	Refs	Inst	DRead	DWrt	Usr	Sys
POPS	3142	1624	1257	261	2817	325
THOR	3222	1456	1398	368	2727	495
PERO	3508	1834	1266	409	3242	266

3 Multiprocessor Model and Definitions

The multiprocessor model we assume for our analyses in this paper is quite straightforward. We assume that the system consists of several processors each with its own cache memory. The caches are connected to a common system bus on which shared main memory is located. We also make the simplifying assumption that caches are infinite in size, since we would like to concentrate on traffic caused due to shared data and not mix it up with traffic due to limited cache size.

We introduce some nomenclature to help explain memory access patterns. A *block* is the unit of data transfer between the cache and main memory. For the rest of the paper, we assume block size to be 1 word (4 bytes). The small block size is chosen so that the reference behavior for each data object can be derived. However, characterization using larger block sizes is also important to study the spatial locality of shared objects, and is dealt with in Section 4.3.

A *read-shared* block is one that is shared (accessed by multiple processors), but never written into. A *write-shared* block is one that is shared, and both read and written into. A reference to a block B by processor i is said to *ping* if the previous reference to that block was by processor j, where $j \neq i$. We call such a reference a pinging reference. Conversely, a reference to a block B by processor i is said to *cling* if the previous reference to that block was also by processor i. Such a reference is called a clinging reference. By these definitions, a ping can only occur on a reference to a shared block. Pings and clings to a block are determined simply by keeping track of which processor last referenced a block. References are read references or write references depending on whether the operation performed is a read or write. The state of a block (clean/dirty) is determined by the references of the processor accessing it currently. A block is said to be dirty if it has been written into after the previous pinging reference to it. Therefore, a block always starts out clean after a pinging reference to it.

The notion of clings and pings yields useful insights on how various shared-memory multiprocessor architectures would perform. The appealing feature of clings and pings is that they do not depend on implementation details such as cache sizes. Assuming a local cache, clinging read references never need the bus: pinging read references need to use the bus only if the read misses or if the block is dirty in another cache. A bus transaction must occur on a pinging write reference. In the ensuing discussion we will show results on the time intervals between such clings and pings, and also on the frequency of various kinds of clings and pings. The time interval plots are a useful method of depicting the temporal locality of shared-memory references, while the frequency of clings and pings is a method of showing the "processor locality" of references to a block. Besides spatial locality and temporal locality, the form of locality important in a multiprocessor context is *processor locality* - the tendency of a processor to access a block repeatedly before an access from another processor. A direct impact of this locality is noticed in the performance of various cache consistency schemes, which exploit different locality patterns in references to read-shared or write-shared blocks. Also notice that a high temporal locality of pinging references yields a low processor locality, and negatively impacts the performance of multiprocessor caches.

To separate the effects of process migration, we also present numbers for *process-migration-shared* blocks. These are blocks accessed from processor i by process p, and also from processor $j \neq i$ by the same process p. On the other hand, *real-shared* blocks, are blocks accessed from processor i by process p, and also from processor $j \neq i$ by process q, where $q \neq p$ always holds.

It is useful to have a notion of time in the context of multiprocessor execution. Our traces contain interleaved memory accesses by the various processors in approximately the same order they occurred. However, the exact time at which the reference was made is not clear. For example, if the processors i, j, and k each made references at real time instants t, $t+1$, and so on, the trace might have references $i_t, j_t, k_t, i_{t+1}, j_{t+1}, k_{t+1}$, and so on, where the order of the t^{th} references of the 4 processors might be random with respect to each other. The traces also show clusters of memory references by the same processor, and the time interval between references by the same processor also varies.

Due to this nature of the reference pattern, we will not try to approximate real time. Instead, we will use the order of occurrence of a reference in the trace as the index of time. So the r^{th} reference in the trace is considered to have occurred at time r.[1] The paper considers several cases where the traces are filtered to extract specific references (e.g., user), and to enable comparisons, the time index used for a reference depends on its index in the original trace. For example, when we filter out operating system references while studying sharing in the user address space, the time index of a user reference corresponds to its position in the unfiltered trace.

4 Results and Analyses

We first present some general statistics about the traces, including data about interlocked instructions. We then present statistics about temporal and processor locality found in the traces when only user references are included and there is no process migration sharing, when both system and user references are included, and when the effects of process migration are taken into account. We then evaluate three different cache coherence schemes on the basis of the amount of traffic they generate on a shared bus. Unless stated otherwise, we assume infinite caches and a 4-byte block size.

4.1 General Statistics

The statistics in Table 2, for both instructions and data references of user and the operating system, relate to the number of unique blocks and the proportion of references to shared blocks in the traces.

Table 2: Proportion of shared references and unique shared blocks when the blocksize is 4 bytes. Both *instruction and data* references of *user and OS* are included. All numbers are in thousands. Block size is 4 bytes.

Trace	Refs	Uniq Blks	Shd Refs	Shd Blks
POPS	3142	37.8	2122	23.7
THOR	3222	78.3	1881	7.0
PERO	3508	22.6	218	4.7

Table 3 gives the same statistics, but only for data references of both user and the operating system. In addition, Table 3 presents the number of blocks that are written. Because the instruction space is usually read-only, it can be treated specially in memory management, and so most of the statistics presented later correspond to data references alone. Table 4 presents the same statistics for user data references alone.

When both user and the operating system data references are considered, the ratio of shared references to all data references (averaged over all three traces) is 0.25; the ratio is 0.27 when only user data references are considered. We see that the level of sharing in the operating system is only slightly lower than in user.

These traces have an insignificant amount of process-migration-related sharing. We also looked at some other traces for the same applications with a large amount of process migration, and the levels of sharing are drastically different in these traces. The ratio of shared to total is 0.9 for user data references when process migration is high; when process migration effects are excluded (only references to real-shared blocks are counted), the ratio of user data references and all data references falls to 0.2.

4.1.1 Statistics for Interlocked Instructions

The VAX architecture provides seven interlocked instructions for synchronization. These are: BBSSI - branch on bit set and set interlocked; BBCCI - branch on bit clear and clear interlocked; ADAWI - add aligned word interlocked; INSQHI, INSQTI, REMQHI, REMQTI - four instructions to manipulate linked lists (queues) in an interlocked manner. The usage of these instructions is presented in Table 5, with separate numbers given for operating system code and user code.

Table 5 shows that only BBSSI and BBCCI instructions occur in the trace. The ADAWI instruction is used in the POPS code, although it does not occur in the instruction references that our trace contains. These statistics show the strong preference of programmers to use the simpler test&set type instructions for synchronization, rather than using the more complex queue manipulation instructions.

The number of interlocked instructions as a fraction of all instruction references is 0.1%-1.6% for the three programs. While the fraction is as high as 1.2%-1.6% for POPS and THOR, the fraction is only 0.1% for PERO. The reason is simply that the author of PERO had made an explicit decision not to use locks for the most frequently used data structure, thus trading the quality of the final solution for extra performance. Since executing an interlocked instruction may be as much as 10-20 times more expensive than an ordinary

[1] We believe that fine time distinctions are not significant in our study. To approximate real time, one can keep a virtual system time incremented by one unit for every n references in the trace, where n is the number of processors. In other words, the times specified in our paper can be divided by 4 to get a rough idea of the real time.

Table 3: Proportion of shared references and unique shared data blocks when the blocksize is 4 bytes. Only *data* references to both *user and OS* are included. All numbers are in thousands.

Trace	Refs	Uniq Blks	Written	Shd Refs	Shd Blks	Shd Wrt
POPS	1518	31.1	9.4	597	19.9	4.0
THOR	1766	74.4	16.6	530	5.2	1.5
PERO	1674	14.0	4.3	136	3.4	0.8

Table 4: Proportion of shared references and unique shared data blocks when the blocksize is one word (4 bytes). Only *data* references of *user* are included. All numbers are in thousands.

Trace	Refs	Uniq Blks	Written	Shd Refs	Shd Blks	Shd Wrt
POPS	1346	29.3	9.1	576	19.8	4.0
THOR	1527	71.9	15.9	473	4.8	1.3
PERO	1528	11.6	3.8	119	3.3	0.73

instruction on some multiprocessors, a small percentage of interlocked instructions can consume a large percentage of total execution time. We also note that most of the interlocked instructions result from the user code and not from the operating system code.

4.2 Temporal and Processor Locality of User Data References

This section deals with dynamic memory access patterns and characterizes the temporal and processor locality of real-shared user data references. The first few figures plot the cumulative distributions and the frequency distributions of the time intervals between clinging and pinging references to demonstrate the temporal locality of data references. All figures use a block size of 4 bytes.

Figure 1(a) shows the cumulative frequency distribution of the time interval between clinging references to a shared block. In other words, a point (x, y) on a curve means that y references occur to a block with the time interval between these references not more than x. The corresponding frequency distribution plot for one of these programs is also shown in Figure 1(b). Due to the wide range of time intervals in which the references occur, the bins on the X-axis increase in powers of two. Therefore a bar at x with height y in the frequency plot, implies that y references occur to a block with an interval t such that $x \leq t < 2x$. For brevity we plot the frequency distributions only for THOR.

The average interval of time between accesses to the same shared block is 1165 time units in THOR. This number is unusually large because even one reference with a very large interval (or an *outlier*) can skew the average towards large values. Therefore, in the context of time intervals, a more interesting number is the median, or the time interval over which half the clinging references occur. It is easy to see that over 50% of the intervals are 25 time units or less in THOR. (The much larger average is due to the bias brought in by a few outliers.) Not surprisingly, these numbers indicate that blocks are re-referenced at small intervals of time, which is simply a reconfirmation of the fact that memory references display a high temporal locality, and is the precise reason why caching is successful. The values at 4K-8K time units form a second peak (Figure 1(b)), although the height is much smaller than the first peak at 16-32 time units. This second peak can be explained as clinging references that occur when the process resumes execution on the same processor after being switched out. The first peak, clearly, is due to references within a context switch interval. The height of the second peak is much larger in traces that show significant process migration. This low temporal locality component of clinging references introduced by process migration can be deleterious to cache performance.

These results are compared with those for pinging references, or for a reference to a block by a processor followed by a reference from another processor. Figure 2(a) shows the cumulative distribution, and Figure 2(b) the frequency distribution. The time intervals in this case are interestingly lower than for clinging references, which says that references to shared blocks by different processors are usually at least as finely interleaved as references by the same processor. Doubtlessly, the fact that our applications exploit parallelism at a fine granularity is the cause of the high temporal locality.

The small second peak at 256 time units in Figure 2(b) is due to the process migrating to another processor following a context switch. If the level of process migration is high, this peak at a large time interval can become much taller, which falsely suggests that process migration lowers the temporal locality of shared references. In reality, process migration simply makes a large fraction of the logically private blocks appear shared, and it is references to these shared blocks alone that give rise to the tall second peak.

Our analysis also shows that roughly a fourth of the data references are to shared data. However, a large part of the shared references need not generate bus traffic because in most multiprocessor architectures, the large number of clinging references to shared blocks (especially reads) can be treated in much the same manner as references to private blocks, in other words, blocks can be treated as private during large windows of time.

The previous figures did not distinguish between read and write references. Making this distinction is necessary because in many high-performance multiprocessor architectures, only pinging references to dirty blocks cause bus traffic when the new value of the dirty block must be somehow transmitted to the requesting processor. Figure 3 shows the distribution of the time interval between pinging references to a dirty block. The total number of pinging references to dirty blocks is far less than all the pinging references. As we shall show later in our discussion on cache consistency performance, sophisticated cache management schemes that take advantage of such features can have significant advantages over simpler schemes.

Table 5: Interlocked instruction statistics. Note the numbers are *not* in thousands.

Trace	BBSSI		BBCCI		OTHERS		TOTAL	
	User	OS	User	OS	User	OS	User+OS	%of-all-Irefs
POPS	9741	302	9375	301	0	0	19719	1.2%
THOR	11619	490	11600	487	0	0	24196	1.6%
PERO	109	437	109	437	0	0	1098	0.1%

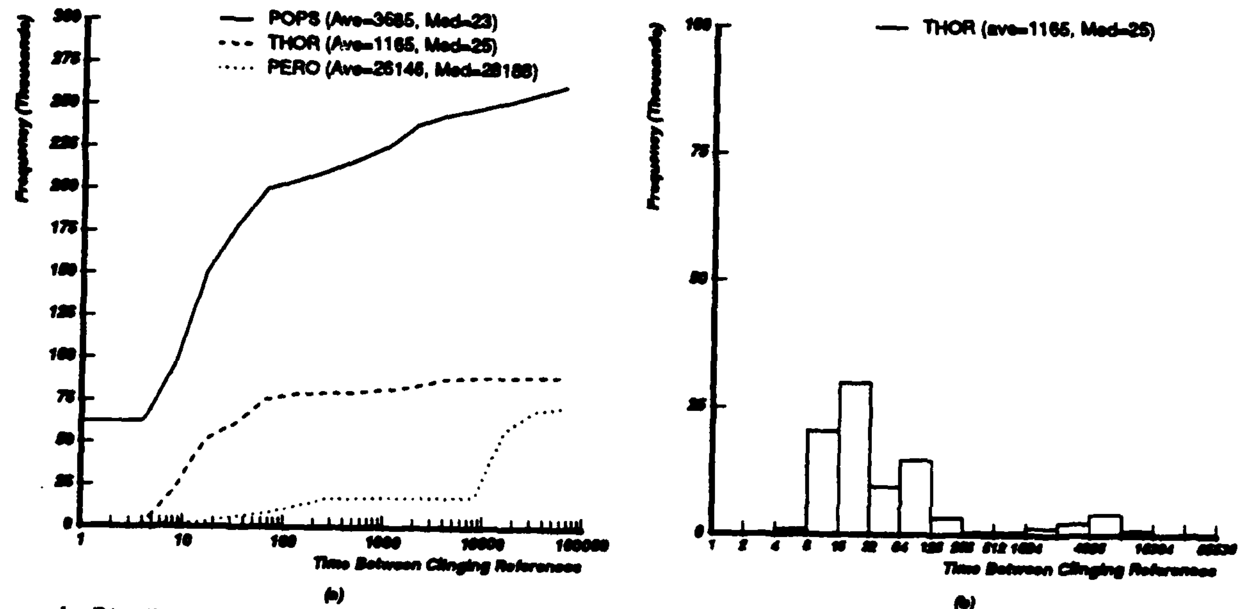


Figure 1: Distribution of the time interval between clinging references to a shared block. Only *real-shared data* references of *user* included.

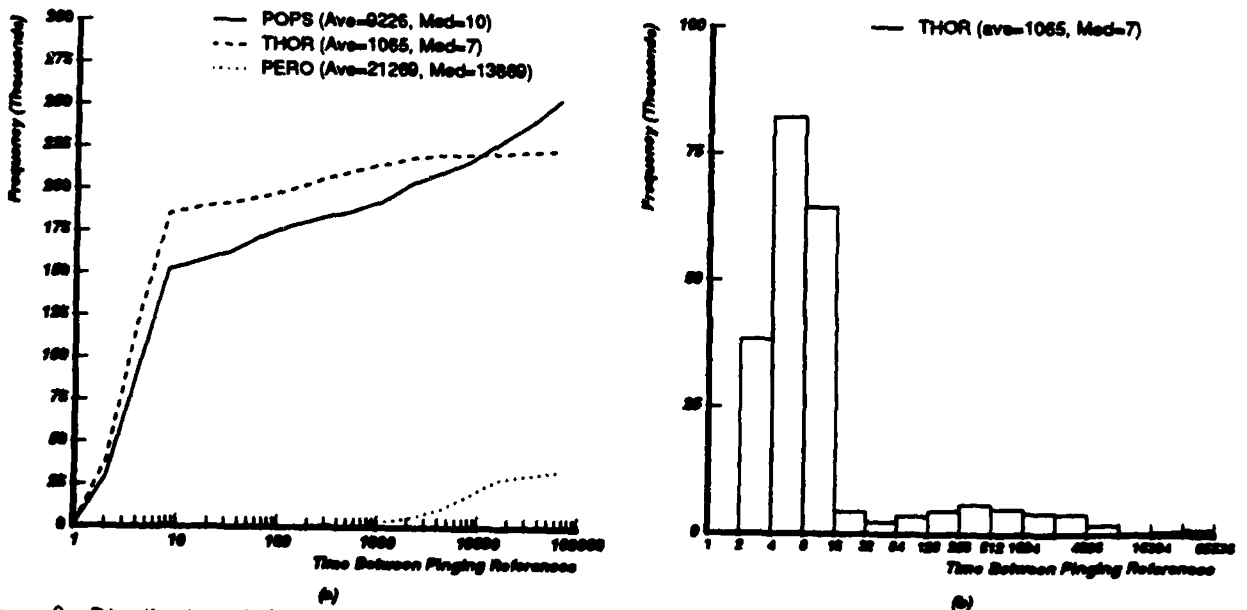


Figure 2: Distribution of the time interval between pinging references to a block. Only *real-shared data* references of *user* included.

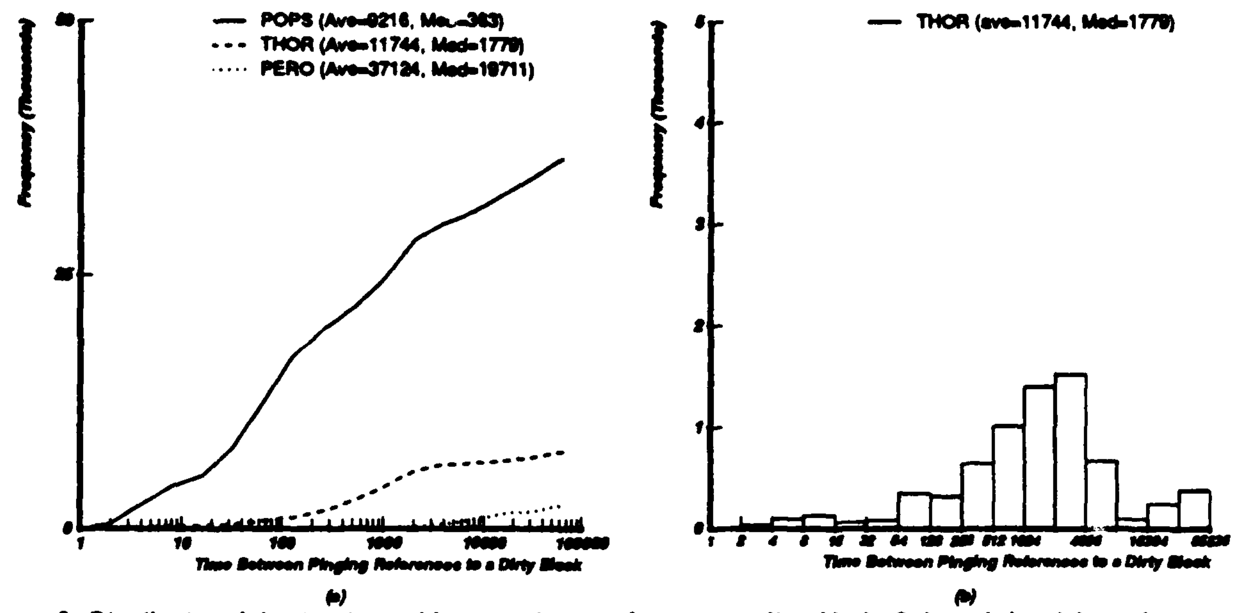


Figure 3: Distribution of the time interval between pinging references to a dirty block. Only *real-shared data* references of user included.

Comparing Figures 2(b) and 3(b), we see that the peak around the time interval 4-8 in Figure 2(b) is caused by reference to read-shared objects. Because Figure 3(b) does not show this early peak, we believe that references to write-shared blocks have less temporal locality than references to read-shared blocks, which benefits multiprocessor caches. A possible case is the test-and-test&set synchronization sequence, where one might expect multiple reads from several processors, but less frequent writes. The low temporal locality in pinging references to dirty blocks encourages us to believe that for large time periods blocks can be considered as private and no traffic need be generated in maintaining consistent caches.

As caches grow bigger, blocks are expected to stay in the cache for long periods of time. In such a situation, a better characterization uses the notion of processor locality. (A similar characterization has also been used in [5]). We will address processor locality in two ways. The first looks at the number of references to a block before a pinging references to it, and the second looks at the number of references to a block before a pinging reference to it, given that at least one of the references was a write. Each of the above two characterizations is pertinent to some cache consistency scheme. For example, the first one indicates the potential of a cache consistency scheme that allows only one cached copy of a block.

Figure 4(a) shows the cumulative distribution of the number of references to a block before a pinging reference, and Figure 4(b) the frequency distribution. In Figure 4(b) for THOR, there are about 200,000 pinging references to a block referenced only once by the previous processor. Unlike in the distributions of time intervals, where we used the median as a measure of temporal locality, here the average is more indicative of processor locality, because outliers represent a large number of references, and must be weighted accordingly. The low average of 1.3 indicates that interleaved references by different processors are as frequent as clinging references, implying low processor locality. We evaluated a cache consistency scheme that allowed only one cached copy of any block [6], and it performed abysmally for this very reason.

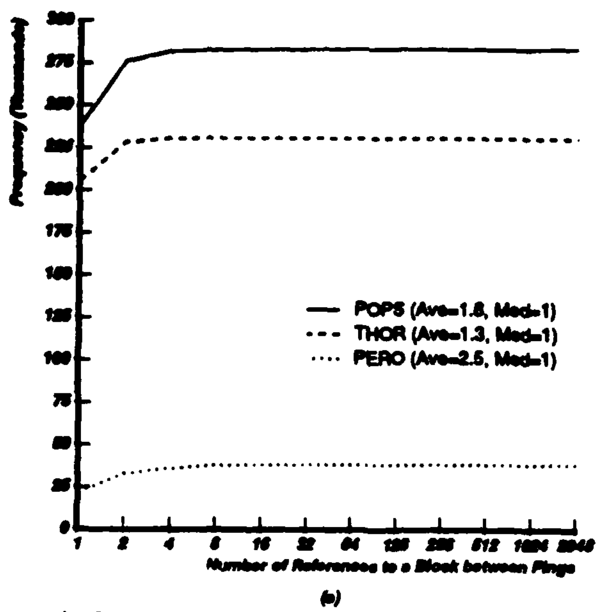
One of the chief differences between some of the snooping cache consistency schemes is the way they treat write references. One set of schemes, e.g., DRAGON [7] or FIREFLY [8], allow caches to hold valid copies of blocks that are being written into by others, and update the values on writes. Another set of schemes prefer to allow only one copy of a written block (e.g., Berkeley Ownership [9], or various flavors of directory schemes [6]). The performance of one or the other method is predicated on the locality of references to write-shared blocks, which we address next.

Figure 5 shows the number of read and write references – at least one reference a write – before a pinging reference. Several observations can be made from this figure. First, the average number of references to write-shared blocks by the same processor before a pinging reference is 5.6 for POPS, 3.6 for THOR, and 7.5 for PERO. Write references are relatively fewer than reads and contribute 1.6, 1.7, and 1.2 respectively to these averages. These averages indicate that the processor locality of shared-writable blocks is higher than that of read-shared blocks. (Recall that the corresponding numbers for all references were 1.8, 1.3, and 2.5). The higher processor locality indicates that a shared written datum is accessed multiple times by a processor before being relinquished.

A more important observation from Figure 5 is that the total number of these pings are approximately an order of magnitude lower than all pinging references, which lessens the adverse impact of the low processor locality of write references on the performance of cache consistency schemes.

As noted earlier, the average number of writes to a block before a pinging reference is small (1.7 for THOR); there are several possible reasons for this low value. We expect a low value for references caused by spinlocks. We also expect this value to be low for shared objects which move from one processor to another, with each processor making some modifications to the object. Also mostly-read-only objects are written once, and then numerous pinging read references are made by other processors.

Thus far, we saw that the processor locality of shared-references is moderate, with roughly 2 writes and 4 reads



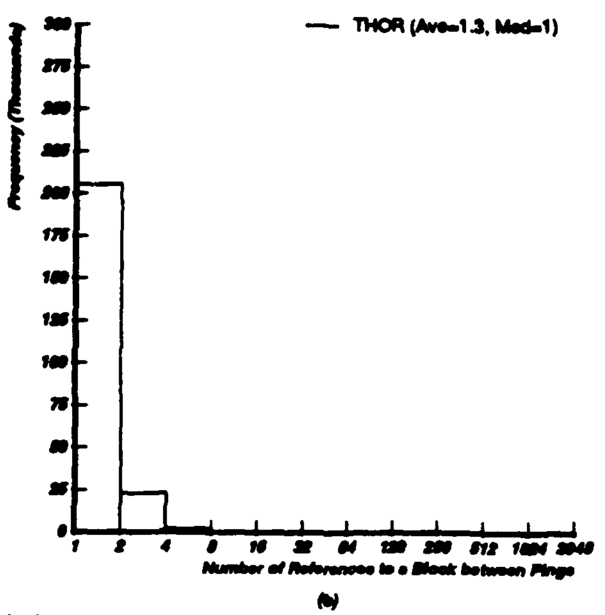


Figure 4: Distribution of the number of references to a block before a pinging reference. Only *real-shared data* references of *user* included.

on average to write-shared objects before a pinging reference. Therefore, a good cache consistency scheme must ensure effective handling of repeat read-references to shared blocks. Given the moderate processor locality of shared-data, we cannot directly determine whether invalidating cache consistency schemes such as the Berkeley Ownership protocol or directory schemes, or the updating protocols such as the Dragon and Firefly schemes are superior. More detailed evaluation that takes into account the cost of updating versus invalidating must be undertaken to make a decision.

4.2.1 Sharing Characteristics of Both User and OS References

The following discussion focuses on the sharing characteristics of both user and system references, where instruction references are excluded, as before. The general observation is that the sharing characteristics of user and system are not significantly different, although the temporal locality of shared system references was slightly lower, and the processor locality was slightly higher.

For the times between clinging references in POPS, THOR, and PERO, the medians occurred at 26, 27, and 27772 for user and system, while the corresponding numbers for user alone were 23, 25, and 28188. The times between pinging references were different by roughly the same ratio, while the times between pinging references to dirty blocks showed greater variation. The medians for user and system were 438, 2095, and 12446, as compared to 363, 1779, and 19711 for user alone.

The processor locality metrics also showed only small differences from the case of user references alone. In general, for the user and system references the average number of references to a block before a pinging reference were roughly 5% greater. A similar trend was observed for the number of references to write-shared blocks.

4.2.2 Effects of Process Migration

Since the three traces we have discussed so far do not show a significant amount of process migration, we used three other traces of the same applications that did. Due to space constraints we will only summarize our findings here and details are presented in [10].

The temporal locality of clinging references decreases if processes are rescheduled on the same processor, after having run on another processor (it will show up as a large increase in the height of the second peak in Figure 1(b)). One component of cache interference caused by migration is similar to the interference caused by context switching.

Perhaps the most important effect of process migration is the significant increase in the number of blocks that get physically shared by several processors, although the logical sharing in the program might be much smaller. For instance, the fraction of references to shared data blocks increases from 0.2 to 0.9 with process migration. Due to the typically long intervals between process switches (thousands of references), the time interval between pinging references to these shared blocks is very large, and causes a much larger second peak in Figure 2(b). Similarly, the average number of references to a block – at least one reference being a write – before a pinging reference is 13 with process migration and less than 2 without. This perceived decrease in the temporal locality and the increase in processor locality of shared references stems from the fact that many of these references are to logically private data objects that are not referenced by other processors until the process actually migrates to another processor.

In summary, although process migration increases the processor locality and decreases the temporal locality of shared blocks, it increases the total number of shared blocks substantially, and potentially impacts both intrinsic cache performance, and the performance of cache consistency schemes adversely.

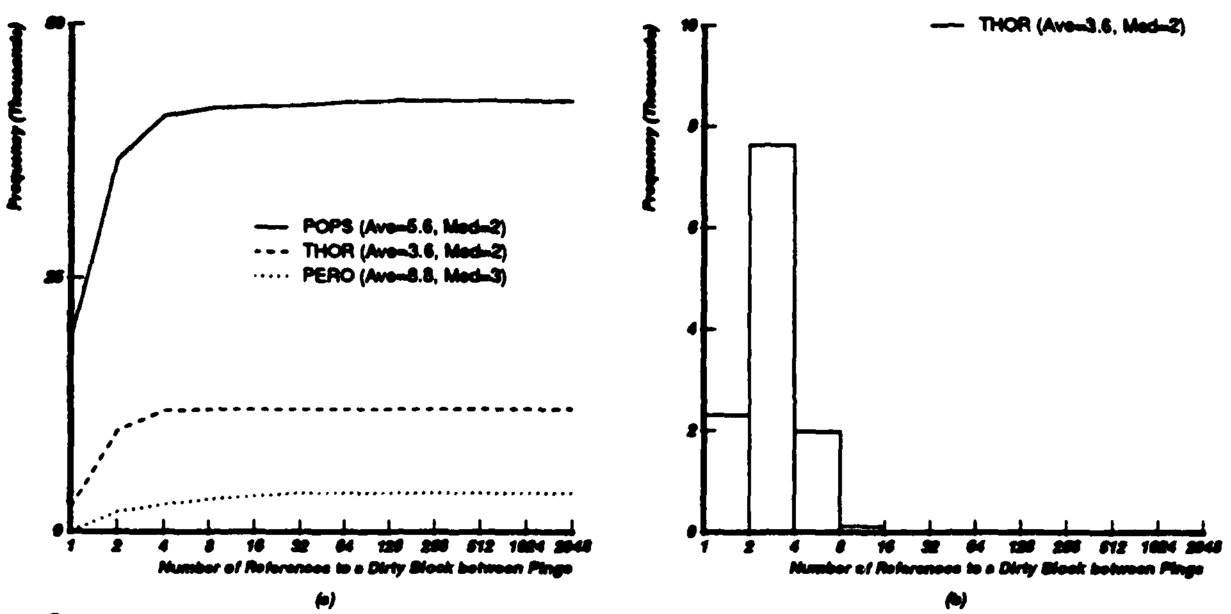


Figure 5: Distribution of the number of references to a block before a pinging reference to the same block, given that at least one reference was a write. Only *real-shared data* references of *user* are included.

4.3 Cache Consistency Implications

The memory reference traces also yield useful insights about the effectiveness of various cache consistency schemes. For example, they enable an accurate determination of the traffic caused on a shared bus by any given cache consistency scheme under realistic load conditions. While a detailed analysis of the numerous cache consistency schemes proposed in literature [11,9,7,12,8] would be interesting, it is beyond the scope of this paper. Instead, we consider one representative each from the write-through with invalidate, write-back with invalidate, and write-back with update classes of cache coherence schemes. To help explain the various phenomena observed here, we use the data presented in earlier sections. As before we assume infinite caches, and unless otherwise stated, block size is one word (or four bytes).

The first scheme discussed in this paper is *write-through with invalidate* (WTI) commonly used in commercial multiprocessors. In this scheme, every write from a processor accesses the bus both to update main memory and to invalidate that location in other caches. Examples of *write-back with invalidate* schemes are Goodman's write-once [11], Rudolph and Segall's scheme [12], Berkeley Ownership [9], and the directory scheme [13]. We consider write-once as the second scheme in this paper. In this scheme, the first write to a location uses the bus to update main memory and to invalidate that location in other caches. Subsequent writes to that location by the same processor do not result in any bus traffic, as that location is now owned locally. This scheme is labeled WBI in the following discussion to indicate the class it belongs to. Examples of the *write-back with update* schemes are Dragon [7] and Firefly [8]. We use Dragon as the third scheme, and denote it WBU. In the Dragon scheme, all writes to a shared location (a location present in multiple caches) result in a bus access to update the value of that location in other caches. For non-shared locations, the cache acts like a regular uniprocessor write-back cache.

We evaluate the performance of the above three cache coherence schemes in terms of the *bus transactions* generated on a shared-memory multiprocessor. We distinguish between three kinds of bus transactions: *block transfers*, *updates*, and *invalidations*. A block transfer transaction transfers a block from memory to cache or vice versa. For example, a block transfer into a cache on a read miss. An update transaction updates the contents of a location either in main memory (e.g., on a processor write in WTI) or in a remote cache (e.g., on a write to a shared location in WBU). The update transfers only one word, and is hence cheaper than a block transfer with a large block size. A processor uses an invalidation to purge cache blocks in other caches to get exclusive ownership of the block. No data transfer is required for this transaction, only the address of the cache block to be invalidated need be specified. Note that block transfers and updates can simultaneously serve as invalidation transactions, and this is usually exploited in most coherence schemes.

Table 6 presents the event frequencies for the three traces as a function of the cache coherence strategy. Because of our interest in characteristics of shared references, we only include cpu-shared user data references for POPS, THOR, and PERO (see Table 4 for details). Because caches are infinite, a data item brought into the cache remains there until invalidated. From Table 6 we derive the total number of block transfer transactions and update transactions that would occur in a multiprocessor and present the numbers in Table 7. The table also presents data for 16-byte and 64-byte blocks to study spatial locality in shared references.

We first examine Table 7 for 4-byte blocks. Comparing total number of transactions, the WTI scheme is worse than both WBI and WBU. WTI looses to WBI because of the processor locality displayed by write references, as shown in Figure 5. While every write generates bus traffic in WTI, clinging write references do not cause bus traffic in WBI. Comparing WTI and WBU, both schemes generate an update transaction for every write to a shared location. However, WBU saves about 25% updates because before the point that a location becomes shared (a second processor requests it), only the first read or write produces a bus transaction. WBU also has fewer block transfers because, unlike WTI, it never invalidates a location from a cache. The details of the events are in Table 6.

Table 6: Events. bus transactions. and event frequencies. Each event is a triple: event-type (read-miss. write-miss. write-hit). state in local cache (not present. clean. dirty), and state in remote cache (not present, cle..n. dirty). We use abbreviations *d* for block transfer. *u* for update. and *i* for invalidate. Only cpu-shared user data references are considered. All numbers are i- thousands.

Event Type	Bus Transactions			POPS			THOR			PERO		
	WTI	WBI	WBU	WTI	WBI	WBU	WTI	WBI	WBU	WTI	WBI	WBU
total refs	-	-	-	575.6	575.6	575.6	473.1	473.1	473.1	119.0	119.0	119.0
read-hits (rh)	-	-	-	429.5	429.5	451.8	416.3	416.3	423.8	102.3	102.3	105.3
read-misses (rm)												
rm-np-np	1*d*	1*d*	1*d*	10.01	10.01	10.01	2.55	2.55	2.55	3.20	3.20	3.20
rm-np-cl	1*d*	1*d*	1*d*	59.24	25.11	13.46	14.21	2.14	0.54	7.13	3.57	2.53
rm-np-di	-	1*d*	1*d*	-	34.13	23.52	-	12.06	6.18	-	3.56	1.57
write-misses (wm)												
wm-np-np	1*d*, 1*u*	1*d*	1*d*	9.72	9.72	9.72	2.28	2.28	2.28	0.08	0.08	0.08
wm-np-cl	1*d*, 1*u*	1*d*	1*d*, 1*u*	12.81	4.39	1.74	0.12	0.01	0.00	0.38	0.10	0.10
wm-np-di	-	1*d*	1*d*, 1*u*	-	8.42	1.90	-	0.11	0.03	-	0.28	0.15
write-hits (wh)												
wh-cl-np	1*u*	1*u*	0	7.64	2.14	2.14	7.39	2.15	2.15	1.38	1.08	1.08
wh-cl-cl	1*u*	1*u*	1*u*	46.63	22.12	1.35	30.27	9.00	0.16	4.53	3.33	1.06
wh-cl-di	-	-	1*u*	-	-	23.87	-	-	8.75	-	-	1.95
wh-di-np	-	0	0	-	30.01	5.50	-	26.50	5.24	-	1.49	0.29
wh-di-cl	-	-	1*u*	-	-	30.58	-	-	21.45	-	-	1.65

Dividing the total number of bus transactions generated by all three programs for the WBI scheme in Table 7 (161.6K) by the total number of references that resulted in these transactions (1168.7K), we see that there are approximately 0.138 bus transactions generated per reference. This number appears quite large given infinite caches, and there are two reasons for this. First, this data represents only cpu-shared user data references, which show poor processor locality as in Figure 4, or equivalently, which display a high temporal locality of pinging references as in Figure 2). Consequently they do not benefit much from the read-sharing allowed by the WBI scheme. If one includes both user and OS references, and both data and instructions, then the number of transactions per reference falls to 0.031, which is much better. This reduction is primarily due to the large number of read-shared references generated by instruction fetches. (Consequently, allowing read sharing for instructions is crucial in multiprocessor caches.) The second reason for the high value is that block size is 4 bytes. When the block size is increased to 16 bytes, the number of transactions per reference drops down further to 0.016, primarily due to the high spatial locality of instruction fetch references.

In general, two opposing forces come into play as the block size is increased – one trying to decrease the number of transactions and the other trying to increase them. As the block size is increased the number of bus transactions is reduced because the bus access or invalidation cost is amortized over several words. Contrarily, a large block size increases the probability of unrelated objects residing in the same block, and a write to one object can unnecessarily invalidate an active unrelated object in a remote cache.

To study the spatial locality characteristics of cpu-shared user data references, we now examine the bus transactions generated by WBI in Table 7 as the block size is increased. For POPS the number of block transfers decreases from 91.86K to 47.15K to 46.23K as the block size is increased from 4 to 16 to 64 bytes. This indicates that there is high spatial locality at 16-bytes, with little cache interference due to coresiding unrelated objects. Beyond 16 bytes, either there is no spatial locality or the cache interference neutralizes the benefits due to locality. THOR behaves differently. When the block size is increased from 4 to 16 bytes, the number of block transfers increases by a factor of 1.5. This indicates that negative cache interference effects dominate.[2] In contrast to POPS and THOR, increasing block size has a very positive effect on PERO. The number of block transfers decrease by a factor of 2 as the block size is increased from 4 to 16 bytes, and further by a factor of 3.4 when the block size is increased from 16 to 64 bytes. The number of update transactions decreases steadily too. Thus the PERO program appears to have high spatial locality with almost no cache interference.

Another interesting result that can be observed by examining the total traffic lines in Table 7 is that for shared data references the total bus bandwidth required is minimized when block size is 4 bytes and increases as the block size is increased. This result is in start contrast to uniprocessor caches, where the optimal block size tends to be much larger. The only exception is the PERO program when block size equals 64 bytes.

We were interested in estimating the effects of obviating broadcasts in cache consistency schemes to enable scalability. Table 8 presents the number of caches in which blocks are actually invalidated, whenever a reference that could potentially invalidate other caches is processed in the WBI scheme. Such references for the WBI scheme are all write misses and all write-hits to a clean location in the local cache. The total number of such references is given in column three. The inv-0 column gives the number of potentially invalidating references that resulted in no actual invalidations, the inv-1 column gives the number of such references that resulted in exactly one invalidation, the inv-2 column gives the number that resulted in an invalidation in two other caches, and the inv-3 column denotes an invalidations in three other caches. Since all the traces are four-processor traces, no reference can result in invalidation in more than three other caches.

[2]Another factor contributing to the increased number of block transfers is the fact that as block size is increased, the number of cpu-shared references also increases.

Table 7: Bus transactions. Only cpu-shared data references of user are included. All numbers are in thousands.

Bus Transactions	POPS			THOR			PERO		
	WTI	WBI	WBU	WTI	WBI	WBU	WTI	WBI	WBU
Block-Size = 4 bytes									
block-xfers (d)	91.86	91.86	60.42	19.15	19.15	11.58	10.79	10.79	7.63
updates (u)	76.79	24.25	59.43	40.06	11.15	30.38	6.37	4.42	4.91
Total Traffic ($d+u$)	168.65	116.11	119.85	59.21	30.30	41.96	17.16	15.21	12.54
Block-Size = 16 bytes									
block-xfers (d)	47.15	47.15	22.97	29.77	29.77	20.27	5.05	5.05	3.44
updates (u)	78.47	15.04	61.48	49.39	12.38	34.02	6.57	2.14	5.26
Total Traffic ($4d+u$)	267.07	203.64	153.36	168.47	131.46	115.10	26.77	22.34	19.02
Block-Size = 64 bytes									
block-xfers (d)	46.23	46.23	9.30	29.75	29.75	16.68	1.50	1.50	0.94
updates (u)	79.39	20.17	65.09	86.99	16.61	73.15	6.95	0.70	5.61
Total Traffic ($16d+u$)	449.23	390.01	139.49	324.99	254.61	206.59	18.95	12.70	13.13

We would like to remark on two aspects of the data presented in Table 8: the fraction of references that invalidate multiple caches as compared to those that invalidate only one cache, and the effect of changing the cache block size. Let us examine the first aspect. The data for 4-byte blocks indicates that the fraction of references that cause invalidations in three caches (1.3%) is quite small compared to the fraction that cause invalidations in one cache (61.0%).[3] It is interesting to speculate if this phenomenon - that on an invalidate transaction, with high probability, data in only one or very few caches needs to be invalidated - is true even when the number of processors is large. If it is true, then instead of building broadcast-based cache consistency mechanisms, one can build message-based mechanisms where the invalidation message is sent only to those caches that contain that data. The resulting reduction in bandwidth requirements makes it possible to build scalable shared-memory multiprocessors. In the following paragraphs, we speculate why the above result should also hold for a larger number of processors.

There are three kinds of data objects in parallel programs: (i) non-shared, (ii) read-shared, and (iii) write-shared objects. The non-shared objects normally do not cause any invalidations except due to process migration, in which case all the invalidations go only to the processor that previously ran that process. The read-shared objects also do not cause any invalidations. So the multiple cache invalidations come from write-shared objects. We now explore some common ways in which write-shared objects are used in parallel programs.

The first common use of write-shared objects is as spin locks or other similar synchronization related structures. Let us consider the spin lock as the typical case. If the spin lock is implemented in a straightforward way using an interlocked test&set instruction, since the instruction ends in a write, at the end of each instruction only one cache contains the data, and only one cache has to be invalidated on a subsequent reference by a different processor. If the spin lock is implemented using a test-and-test&set instruction,[4] then with some probability the lock will be present in multiple caches. When the lock is set free by writing into it, these multiple caches have to be invalidated. However, if the program is "reasonable" (i.e., there is no excessive contention for the locked object), then either the lock will not have too many processes waiting on it and thus only one or a few caches will need to be invalidated, or such an occurrence will be very rare, and the probability of invalidating many caches will be very small.

The second common use of write shared objects is as mostly-read-only objects. An example is multiple programs sharing a database that is occasionally modified. By occasionally we mean that relative to the number of references made to that object, the number of writes is small. On a write to a mostly-read-only object, multiple caches may have to be invalidated, but since writes are rare, the overall fraction of multiple cache invalidations still stays low. The third common use of write-shared objects is where one process works on an object for some time, then another process, and so on. Shared objects protected by locks often behave this way. In this third case, when one process is working on an object, that object resides in the cache of the associated processor. When that object moves to another process (and possibly to another processor), the cache entries in the previous processor are invalidated, but that corresponds to invalidation in only one other cache. So it is still consistent with our conjecture that in larger multiprocessors invalidations will happen in only one or in a very small number of other caches with high probability. The above observations suggest the use of a message-based cache consistency protocol, instead of a broadcast-based protocol. We are analyzing this issue in detail and results will be presented in a future paper.

We now look at the effect of increasing the cache block size on the number of invalidations. The fraction of references that cause invalidations in multiple caches increases with block size. As an example, for POPS, consider dividing the entries in the inv-3 column by corresponding entries in the total column in Table 8. The numbers we get are 2.1%, 4.6%, and 6.2% respectively. The primary reason for this phenomenon is that as block size is increased, unrelated data objects fall into the same cache block. Multiple processors accessing these distinct objects cache the same block, and a subsequent write results in an invalidation in multiple caches.

5 Summary and Conclusions

We have presented data characterizing the memory reference patterns in shared-memory multiprocessors. Our data is based on traces obtained for three applications from a 4-

[3] The reason why this ratio is smaller for POPS and THOR for larger block sizes is discussed later.

[4] In a test-and-test&set instruction, if the first test fails we simply loop back and do not execute the test&set part of the instruction.

Table 8: Cache invalidation statistics for the WBI coherence scheme. Only user cpu-shared data references are included. All numbers are in thousands.

Trace	B	total	inv-0	inv-1	inv-2	inv-3
POPS	4	46.77	11.85	29.24	4.69	0.99
	16	27.06	3.89	18.51	3.42	1.24
	64	30.18	1.33	20.07	6.92	1.86
THOR	4	13.55	4.43	8.97	0.13	0.02
	16	14.72	5.11	8.69	0.74	0.18
	64	18.06	3.28	13.72	0.94	0.12
PERO	4	4.87	1.16	2.65	0.98	0.08
	16	2.18	0.47	1.17	0.50	0.04
	64	0.72	0.14	0.42	0.14	0.02

processor VAX 8350 using the ATUM address tracing technique. The traces used are "complete", in that they contain information about both system and user references, references due to interrupts, process scheduling, etc.

Our analyses shows that a large fraction (about one-fourth) of references in the traces are to shared objects. These shared references display a significant amount of temporal locality, and only a small amount of processor locality for both read and write references. For example, the average number of reads and writes to a write-shared block before a remote reference (a ping, which may possibly invalidate the data) are 4 and 2 respectively. Nevertheless, caching shared data is still highly useful because of the significant amount of read sharing.

We also present statistics about the use of interlocked instructions. The traces show that 0.1%-1.6% of instruction references are to interlocked instructions, and that most of these instructions references are from user code. The paper also touches on the effects of process migration. Process migration causes a large number of logically unshared references to become shared references with respect to the cache system.

The nature of shared-memory reference patterns also yields insight on how various cache consistency schemes will perform. We present the analysis for three classes of cache consistency schemes – write-through with invalidate (WTI), write-back with invalidate (WBI), and write-back with update (WBU). For shared data references, WTI performs worse than both WBI and WBU as it uses the bus on every write. Comparing WBI and WBU, the former seems to have an edge for 4-byte blocks, while WBU does better for 16-byte and 64-byte blocks. Another surprising result that we observed for shared data references is that the total bus bandwidth required is minimized when block size is 4 bytes and increases as the block size is increased. Our traces also show that when a reference that could possibly invalidate a cache is processed, with a very high probability (61.0 %) it invalidates only one other cache. The probability of causing an invalidation in all three caches is only 1.3%. We discuss why this should also be true for multiprocessors with larger number of processors, and suggest the use of message-based cache consistency schemes rather than broadcast-based cache consistency schemes.

6 Acknowledgements

Several people have helped us in obtaining the traces. We to thank Roberto Bisiani and the Speech Group at CMU for letting us use their VAX 8350. Dick Sites at Digital Equipment Corporation, Hudson, made multiprocessor ATUM possible, and Digital Equipment Corporation made the ATUM microcode available for our use. Larry Soule and Helen Davis at Stanford helped with the THOR program and Jonathan Rose with PERO. Finally, many ideas presented in this paper came up during discussions with Susan Eggers, Mark Horowitz, John Hennessy, and Rich Simoni. We appreciate their contributions. The research reported in this paper was funded by DARPA contract MDA903-83-C-0335. Anoop Gupta is also supported by a faculty development award from DEC.

References

[1] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–127, June 1986.

[2] F. Darema-Rogers, G. F. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 46–58, May 1987.

[3] Anoop Gupta, Charles Forgy, and Robert Wedig. Parallel architectures and algorithms for rule-based systems. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, June 1986.

[4] Jonathan Rose. *LocusRoute: A Parallel Global Router for Standard Cells*. Technical Report, Computer Systems Laboratory, Stanford University, 1987.

[5] Susan J. Eggers and Randy H. Katz. *A Characterization of Sharing in Parallel Programs and its applicability to Cohe·ency Protocol Evaluation*. EECS Department, UC Berkeley, October 1987.

[6] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. *Scalable Directory Schemes for Cache Coherence*. Computer Systems Laboratory, Stanford University, October 1987. Submitted for publication.

[7] E. McCreight. *The Dragon Computer System: An Early Overview*. Technical Report, Xerox Corp., September 1984.

[8] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[9] R. H. Katz et al. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[10] Anant Agarwal and Anoop Gupta. *Memory-Reference Characteristics of Multiprocessor Applications under MACH*. Computer Systems Laboratory, Stanford University, February 1988.

[11] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, June 1983.

[12] L. Rudolph and Z. Segall. Dynamic decentralized cache consistency schemes for mimd parallel processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 340–347, June 1985.

[13] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, c-27(12):1112–1118, Dec. 1978.